Documentation for Alevin Developers

Last update
May 24, 2013

Contents

1 Adding new algorithms

1.1 Basic Structure

- Any algorithm used with VNREAL must be derived from MuLaViTo's `mulavito.IAlgorithm` interface which
  - defines a common way to access status information of a running algorithm
  - allows to show a GUI progress bar
- The package *vnreal.algorithms* contains several base classes (implementing `mulavito.IAlgorithm`) to derive own algorithms from
- `vnreal.algorithms.AbstractAlgorithm` (shown left in figure 1) merely provides abstract methods for doing stuff before and after running the algorithm
- `vnreal.algorithms.AbstractSequentialAlgorithm` (shown right in figure 1) performs a sequential processing providing abstract methods *hasNext* and *getNext*

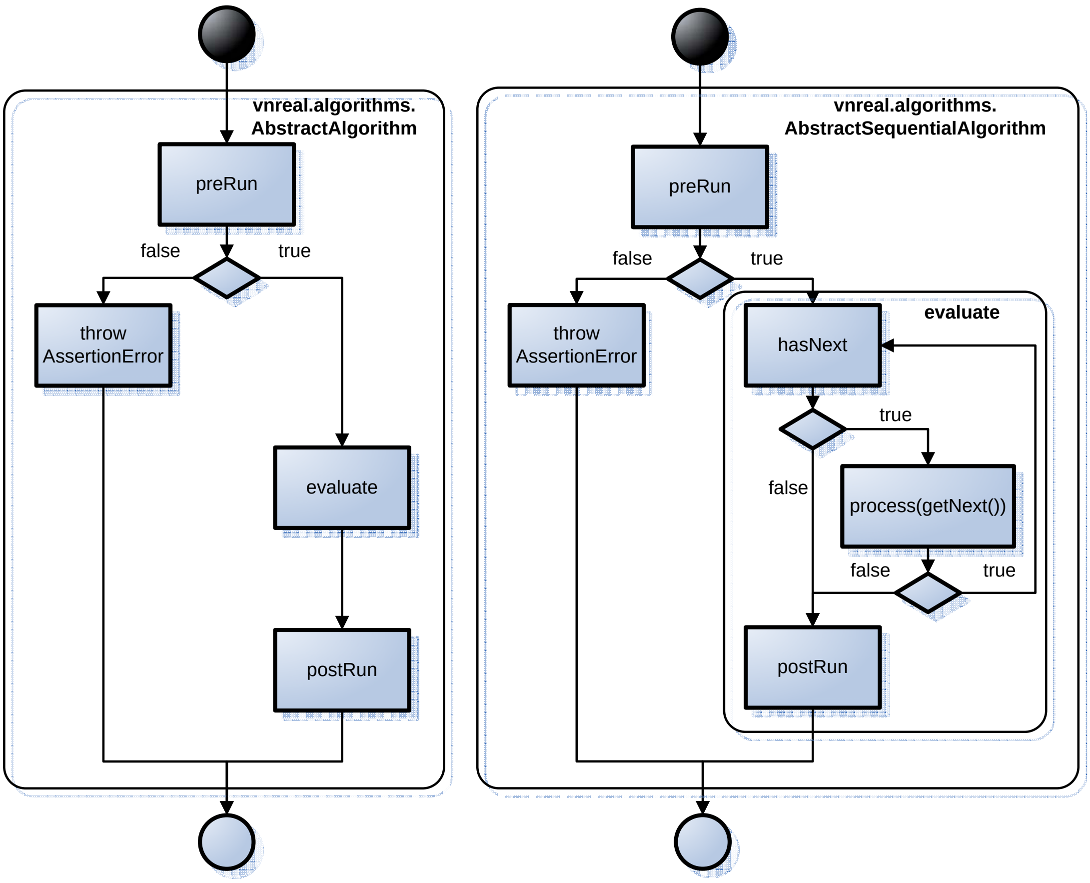

Figure 1: Behaviour of `AbstractAlgorithm` and `AbstractSquentialAlgorithm`

- `vnreal.algorithms.AbstractRevokableSequentialAlgorithm` additionally provides an abstract *revoke* method

- Basic algorithmic principle
  1. The algorithm gets a `vnreal.network.NetworkStack` which consists of
     - a `vnreal.network.substrate.SubstrateNetwork` with resources
     - a list of `vnreal.network.virtual.VirtualNetwork` with demands
  2. The algorithm performs the VNM and VLM by search for resources that are able to fulfill the given demands

1.2 Example

The package *vnreal.algorithms.samples* contains experimental algorithms, like the `SimpleDijkstraAlgorithm`. This exemplary algorithm performs

- an arbitrary virtual node mapping (choosing the node mapping in an arbitrary way among the substrate nodes accomplishing the virtual node demands),
- the virtual link mapping is implemented by connecting each pair of mapped virtual nodes by the shortest path in the substrate network calculated using the Dijkstra algorithm.

The first part of the class `SimpleDijkstraAlgorithm` implementation looks like

```
public final class SimpleDijkstraAlgorithm extends
    AbstractSequentialAlgorithm<VirtualLink> {
  private final NetworkStack stack; //Set of networks including
      the substrate network and the set of virtual network
      requests.
  private Iterator<? extends Network<?, ?, ?>> curNetIt; //
      Iterator over the set of virtual network requests
  private Iterator<VirtualLink> curIt; //Interator over the set
      of virtual links of a virtual network request
```

This is the *hasNext* function of this algorithm which simply updates the iterator over the set of virtual network requests. If one virtual network requesthas already been served, it moves the *curNetIt* to the next virtual network request and updates *curNetIt* iterator. If there is no more virtual network requests it returns false.

```
@Override
protected boolean hasNext() {
  if (curIt == null || !curIt.hasNext()) {
    if (curNetIt.hasNext()) {
      Network<?, ?, ?> tmp = curNetIt.next();
      if (tmp instanceof SubstrateNetwork)
        tmp = curNetIt.next();
      curIt = ((VirtualNetwork) tmp).getEdges().iterator();
      return hasNext();
```

```
    } else
      return false;
  } else
    return true;
}
```

The *getNext* method returns the following virtual link that will be mapped in the process method. To see the example code in detail take a look at the *vnreal.algorithms.samples* package.

```
@Override
protected VirtualLink getNext() {
  if (!hasNext())
    return null;
  else
    return curIt.next();
}
```

2 Easily Reusing Existing Node/Link Mapping Algorithms

The `SimpleDijkstraAlgorithm` is an example of how to implement an algorithm to solve the virtual network mapping, when the virtual link and node mapping are performed as a single stage. However, most of the algorithms to solve the VNE are divided in two stages:

1. Virtual node mapping (performed in first place) and
2. Virtual link mapping (performed in second place).

To facilitate the task of implementing a VNE algorithm, the abstract `vnreal.algorithms.GenericMappingAlgorithm` class should be used.

The `GenericMappingAlgorithm` class takes two parameters: NodeMapping and LinkMapping. These need to be derived from the two classes `vnreal.algorithms.AbstractNodeMapping` and `vnreal.algorithms.AbstractLinkMapping`, respectively. In this way, the node mapping and link mapping stages can be implemented independently of each other. This has an important advantage: the node and link mapping stages of different algorithms can be combined, which may leads to different results.

GenericMappingAlgorithm works by processing each time a different virtual network request so the *getNext()* method returns a virtual network request. The *process(VirtualNetwork p)* method, receive this virtual network request and performs node mapping and link mapping stages. If at least one of them were not successful, all the mappings over that virtual network are undone and the method finishes (without realizing the mapping of that network); if they were both successful, the method finishes. The flow chart in figure 2 of the algorithm is shown next. Different methods of `AbstractNodeMapping` and `AbstractLinkMapping` are described after the figure.

To implement a new algorithm that has both, node and link mapping stage, two classes need to be implemented. Node mapping class that extends from `AbstractNodeMapping` and the link mapping class that extends from `AbstractLinkMapping`.

2.1 AbstractNodeMapping

Now let's check the **methods and variables** of `AbstractNodeMapping`:

```
public abstract class AbstractNodeMapping {
  protected Map<VirtualNode, SubstrateNode> nodeMapping;
  private List<VirtualNode> unmappedvNodes;
  private List<SubstrateNode> unmappedsNodes;
  protected List<SubstrateNode> mappedsNodes;
```

The previous global variables have the following meaning:

- *nodeMapping*: It is a list of Map type that contains the mapping of each `VirtualNode` to its corresponding `SubstrateNode` after the node mapping is realized. This variable should be updated while mapping is being performed.

- *unmappedvNodes*: It is a list with the virtual nodes that have not been mapped for the current virtual network request, after a predefined node mapping have been performed. ALEVIN supports the possiblity of predefining the mapping of a set of virtual nodes (possibly all) into its corresponding substrate nodes.
- *unmappedsNodes*: It is a list with the substrate nodes that have not been mapped, for the current virtual network request, after a predefined node mapping have been performed.
- *mappedsNodes*: It is a list with the substrate nodes that have been mapped, for the current virtual network request, performing the predefined node mapping.

```
public boolean isPreNodeMappingFeasible(VirtualNetwork vNet)
```

ALEVIN supports a resource/demand called id in nodes: In this way it is possible to realize a predefined node mapping before the algorithm runs, it is enough to assign in the `IdDemand` of the virtual nodes the `IdResource` of the substrate nodes that are chosen to realize the mapping. The *isPreNodeMappingFeasible* method is responsible of realizing this predefined mapping and ensures that the resources of the mapped substrate nodes are enough to cover the demands of the virtual nodes. The method returns a false value if the mapping could not be performed. This method is already implemented in `AbstractNodeMapping` and used in `GenericMappingAlgorithm`. After the method is performed, *nodeMapping*, *unmappedvNodes*, *unmappedsNodes* and *mappedsNodes* are updated.

```
public boolean isPreNodeMappingComplete() {
  return unmappedvNodes.isEmpty();
}
```

The method *isPreNodeMappingComplete()* checks if in the predefined node mapping stage the virtual node mapping has been performed completely or not. It is complete if all virtual nodes have been mapped, in this case no virtual node mapping should be performed. This method is already implemented in `AbstractNodeMapping` and used in `GenericMappingAlgorithm`.

```
protected abstract boolean nodeMapping(VirtualNetwork vNet);
```

The *nodeMapping* method is the most important method of the class. **This method** is the one that has to be implemented in a new class (extending `AbstractNodeMapping`). It is important to know, when implementing this method, that probably a predefined mapping has been performed and some virtual nodes of the virtual network request have already been mapped (take into account *nodeMapping*, *unmappedvNodes*, *unmappedsNodes* and *mappedsNodes* variables). After the *nodeMapping* has been performed, the *nodeMapping* variable should be updated and the method should return a boolean value (true if the node mapping was successful, false otherwise). To see some implementations of the *nodeMapping()* method, please go to *vnreal.algorithms.nodemapping* package.

2.2 AbstractLinkMapping

Now let's check the **methods and variables** of `AbstractLinkMapping`:

```
public abstract class AbstractLinkMapping {
  protected int processedLinks, mappedLinks;
```

The `AbstractLinkMapping` variables *processedLinks* and *mappedLinks* are used to update the progress bar of the algorithm. When a virtual link is being mapped, *processedLinks* should be incremented by 1, and when it have been already mapped, *mappedLinks* should also be incremented by 1.

```
protected abstract boolean linkMapping(VirtualNetwork vNet,Map<
    VirtualNode, SubstrateNode> nodeMapping);
```

The main method of the `AbstractLinkMapping` class is the abstract *linkMapping* method. **This method** is the one that has to be implemented in a new class (extending `AbstractLinkMapping`). The input is the virtual network request and the already performed **node mapping**. The output of the method should be a boolean true if the node mapping was successful, false otherwise).

2.3 Required GUI adoptions

If you want your new algorithm to appear in the GUI, simply extend the class `vnreal.gui.menu.AlgorithmsMenu`.

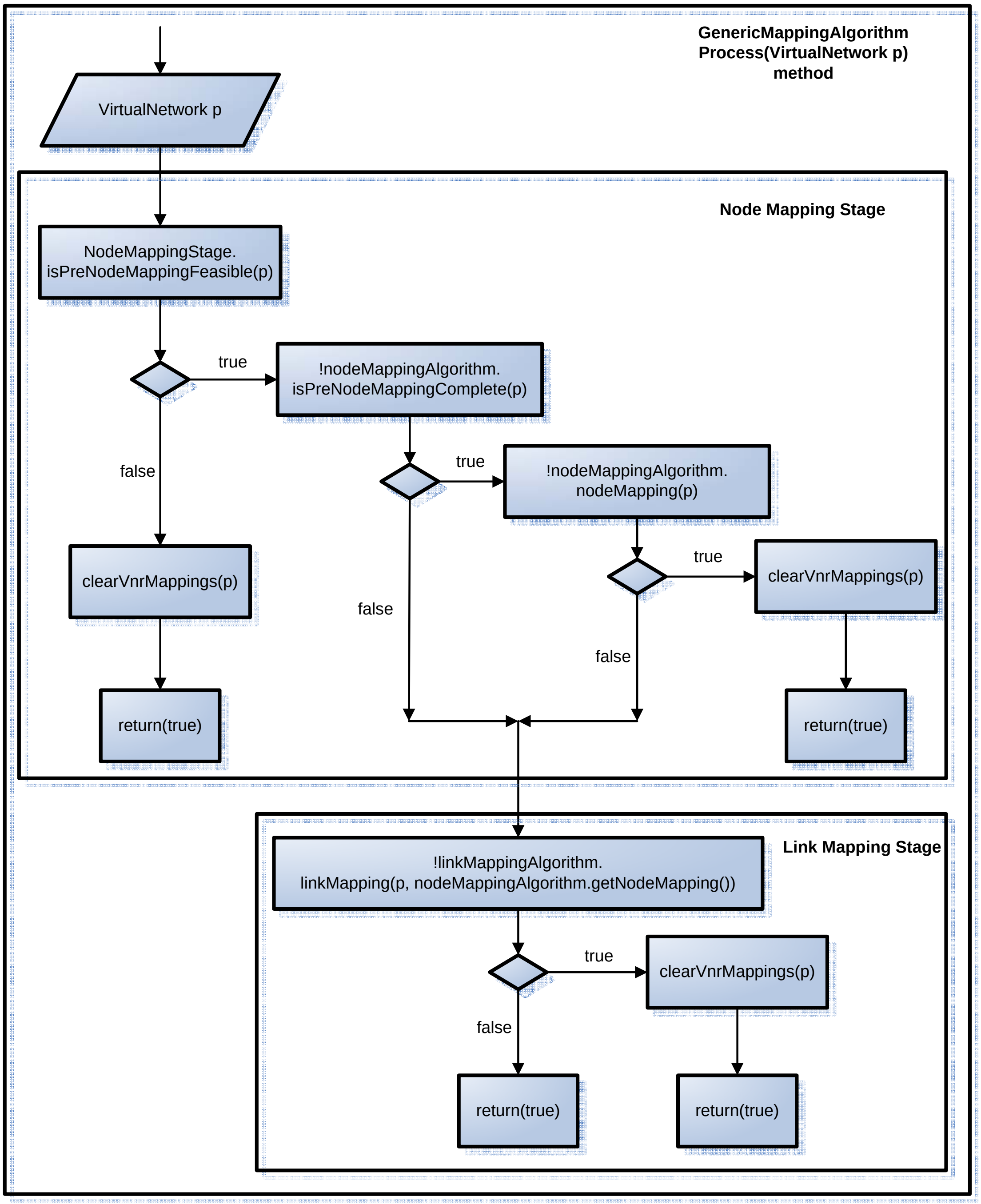

Figure 2: Behaviour of `GenericMappingAlgorithm`

3 Adding of New Pair of Resource/Demand Types

3.1 Overview

- All resource types are derived from `AbstractResource`
- Resources may only be added to the entities of the `SubstrateNetwork`
- The `SubstrateNetwork` consists of `SubstrateLinks` and `SubstrateNodes`
- All demand types are derived from `AbstractDemand`
- Demands may only be added to the entities of a `VirtualNetwork`
- A `VirtualNetwork` consists of `VirtualLinks` and `VirtualNodes`

We use the **Visitor pattern**[1] and **Adapter pattern**[2] to avoid

- casts to concrete demand/resource classes,
- the use of Java's instanceof as far as possible.

N.b.: We use these design pattern in an entangled way.

- A resource is a visitor for a demand providing the occupy and free visitors
- A demand is a visitor for a resource providing the accepts and fulfills visitors
- Do not get confused about this.
- Every Resource and Demand has a name. When you create a clone (with *getCopy()*) you have to copy the name over to the clone. So that the equals()-Method can do matches with clones.
- If you don't provide a name, the name is set to **NameOfOwner_this.hashCode()**

3.2 About the visitor pattern

Normally the visitor pattern is used to be able to add new operations to an existing object structure, without altering every single object. The operations get encapsulated in visitors, which then visit the objects and interacts with them. The object only have to provide a simple interface to accept visitors. Here we already know all operations we need (free, occupy, accepts, fulfills), but want to be able to add new resource / demand pairs to our network, e.g. new objects, which then can handle all upcoming resource / demand requests.

[1] See http://en.wikipedia.org/wiki/Visitor_pattern
[2] See http://en.wikipedia.org/wiki/Adapter_pattern

3.3 Constraints

For ALEVIN's **XML exchange format**[3], resource and demand classes need to meet the following constraints

- The constraint classes must implement one or both of the `INodeConstraint`, `ILinkConstraint` interfaces. This shows whether the Constraint is applicable to nodes, links or both.
- For each parameter that shall be included in the exchange format, a getter and a setter method **must be declared and annotated** with @ExchangeParameter
  - Getters are required for export
  - Setters are required for import
- The parameters for these methods **must not** be simple types (int, double) but classes such that it can be used via Java Reflection
  - N.B. Currently, the exchange format supports the following types: Integer, Double, String, Boolean and ArrayList<String>
- Resources
  - Getter methods must be named according to the following pattern: get + <parameter name>
  - Setter methods must be named according to the following pattern: set + <parameter name>
- Demands
  - Getter methods must be named according to the following pattern: getDemanded + <parameter name>
  - Setter methods must be named according to the following pattern: setDemanded + <parameter name>
- Constructors
  - It's suggested that you use a default constructor only with the owner as parameter.
  - If you need to have additional parameter you need to add the @ConstructionParameter annotation
  - If you have more than one parameter, the parameterName and parameterGetter have to be in the right order.

```
@AdditionalConstructParameter(
    parameterNames = {"sNetwork"},
    parameterGetters = {"getsNetwork"}
)
```

[3]See SVN_base/src/XML/Alevin.xsd

```
public final class IdResource extends AbstractResource
    implements
     INodeConstraint {
  private String id;
  private final SubstrateNetwork sNetwork;


  public IdResource(Node<? extends AbstractConstraint>
      owner, SubstrateNetwork sNetwork) {
    super(owner);
    this.sNetwork = sNetwork;
  }
  ...
```

3.4 Procedure

To add the new demand type `MyNewDemand` and the corresponding resource type `MyNewResource` perform the following steps:

1. Add dummy methods to
   - *vnreal.demands.DemandVisitorAdapter*

     ```
     public boolean visit(MyNewDemand req) {
       return false;
     }
     ```

   - *vnreal.resources.ResourceVisitorAdapter*

     ```
     public boolean visit(MyNewResource req) {
       return false;
     }
     ```

2. Create a new class `MyNewResource` in package *vnreal.resources* extending `AbstractResource`
3. Create a new class `MyNewDemand` in package *vnreal.demands* extending `AbstractDemand`
4. Implement all abstract methods (imitate existing code, like `IdDemand` and `IdResource`)

4 Mapping of Demands on Resources

4.1 How Demands are Mapped to Resources and Vice Versa

Mappings between demands and resources are established by the mapping algorithms. For each demand, the algorithms determine the resources fulfilling it.

To create a mapping between a demand an a resource the following steps ar needed:

1. The resource must accept the demand. In this step, it is determined if the resource and demand are compatible.

2. The resource must fulfill the demand. This assures that the resource is sufficient for the demand's requirements.

3. Finally, the demand occupies the resource. In this step, the free capacity of the resource is reduced and a mapping between the two constraints is created.

To remove an existing mapping, the demand must free the occupied resource.

4.2 How to Deal with Demand-Resource Mappings

- Access
  - Get occupied resource from a demand:

```
AbstractDemand d;
for (vnreal.mapping.Mapping m : d.getMappings()) {
  AbstractResource r = m.getResource();

  // ...
}
```

  - Get occupying demands of a resource:

```
AbstractResource r;
for (vnreal.mapping.Mapping m : r.getMappings()) {
  AbstractDemand d = m.getDemand();

  // ...
}
```

  - Removal

```
AbstractDemand d;
AbstractResource r;

d.getMapping(r).unregister(); // unlinks both elements
```

4.3 Example for occupying and freeing resources

To occupy a resource for a virtual network, the *occupy()* method of the corresponding Demand gets called. The *occupy()* method retrieves the occupy visitor from the Resource, which then visits the Resource.

First it checks if the Resource is able to fulfill the Demand by calling the *fulfills()* method of the Resource. The *fulfills()* method retrieves the fulfills visitor from the Demand, which then visits the Resource and returns if it is able to fulfill the demand. This result gets forwarded to the occupy visitor, which then creates a new Mapping to occupy the demanded resource. The Mapping registers itself at the Demand and the Resource.

At last the occupy visitor returns if the resource occupation was successful. This results gets forwarded to the caller of the *occupy()* method.

Missing Pucture... occupy pdf

To free a resource, which is no longer needed, the *free()* method of the corresponding Demand gets called. The *free()* method retrieves the free visitor from the Resource, which then visits the Resource.

The free visitor takes the corresponding mapping and advises it to unregister itself from the Demand and the Resource. If this was successful, the Resource is freed and the result gets forwarded to the Demand and from there to the caller of the *free()* method.

Missing Pucture... free pdf

5 Test Generation, Running and Plotting

5.1 The Basic concept

The test-system uses the classes `TestRun` and `TestSeries`, which can be found in package *test*, as the basis for handling the data. A `TestRun` represents one test with the parameters, results and the complete `Scenario`. The `TestSeries` consists of at least one `TestRun`, a name and the "TestGenerator" which created the tests.

The test-system consists of two parts. The first part includes classes which are generating tests with parameters or working with them. This category contains the `TestRun`, `TestSeries`, `XMLImporter`, `XMLExporter`, `PlainFileExporter`, `Plotter` and the "TestGenerators".

A "TestGenerator" which is an implementation of `AbstractTestGenerator`, creates the `TestSeries` with all necessary `TestRuns`. The implementation sets the class of the "TestRunner" which will run the Tests, name of the test-series, class of the generator and the number runs for one distinct parameter set. For use in parameters the types `Double` and `String` are allowed. The followig arrays for are already predefined in `AbstractTestGenerator`:

```
public static final Double[] numSNodesArray = {100d};
public static final Double[] numVNetsArray = {5d};
protected static final Double[] numVNodesPerVNetArray = {10d};
public static final Double[] alphaArray = {0.5};
public static final Double[] betaArray = {0.5};
```

If it's necessary, it's possible to redefine them.

The second part contains classes which execute the tests. A "TestRunner" runs the different `TestRun`(s) contained in the given `TestSeries`. The implementation of a "TestRunner" sets the "SeedGenerator", "NetworkGenerator" and "ResourceGenerator(s)" and "DemandGenerator(s)". All these generators may need to contain different parameters, if a `TestRun` is lacking a necessary parameter an error is thrown. The "SeedGenerator" is optional, but some other generators rely on it, which leads to an error. The "TestRunner" may run tests in a given number of parallel running threads, this may lead to wrong results, because some algorithms are not thread-safe and using static data structures.

UML Missing here

5.2 Running Tests

To run a test, it's necessary to create a `TestSeries`. This can be done by using a "TestGenerator" or by importing an existing `TestSeries` using the `XMLImporter`.

When using a "TestGenerator" it's necessary to create an implementation of the `AbstractTestGenerator`. In the constructor there is a need to call the super-constructor and initialize the list of parameters *mParams*. Each entry contains a name and an array of values, which can have `Double` or `String` as data type. The name may contain a "marker", in this case "S1" (see exmplate below). If a name contains a marker these parameter will be synchronized which means they will be rotated together in one step.

```java
public class ExampleTestGenerator extends AbstractTestGenerator
    {
  protected static final Double[] numVNodesPerVNetArray = { 5d,
      10d };
  public static final Double[] numVNetsArray = { 20d, 4d };
  public static final Double[] kShortestPath = { 3d };
  public static final Double[] cpu_min = { 10d, 100d };
  public static final Double[] cpu_max = { 100d, 200d };

  public ExampleTestGenerator(String seriesName) {
    super(ExampleTestRunner.class, seriesName, "tests.scenarios
       .example.ExampleTestGenerator", 30);

    mParams = new ArrayList<SimpleEntry<String, Object[]>>();
    //S1 is a marker, which lets these parameters treated in
       one round
    mParams.add(new SimpleEntry<String, Object[]>("Waxman_alpha
       ", alphaArray));
    mParams.add(new SimpleEntry<String, Object[]>("Waxman_beta"
       , betaArray));
    mParams.add(new SimpleEntry<String, Object[]>("SNetSize",
       numSNodesArray));
    mParams.add(new SimpleEntry<String, Object[]>("
       NumVNodesPerNet", numVNodesPerVNetArray));
    mParams.add(new SimpleEntry<String, Object[]>("NumVNets",
       numVNetsArray));
    //S1 is a marker, which lets these parameters treated in
       one round: (min,max) := (10,100), (100,200)
    mParams.add(new SimpleEntry<String, Object[]>("S1:Min_CPU",
        cpu_min)); //For CPU generators
    mParams.add(new SimpleEntry<String, Object[]>("S1:Max_CPU",
        cpu_max)); //For CPU generators
  }
}
```

Now it is possible to create an object of the `ExampleTestGenerator` and generate the `TestSeries` calling the method *generateTests()*. In the example 240 `TestRun` objects are created, for any possible combination of parameters, 8 in this case with 30 distinct runs for each.

The next step is to run the tests, it is necessary to implement a "TestRunner" using the `AbstractTestRunner`.

```java
public class ExampleTestRunner extends AbstractTestRunner {
  public ExampleTestRunner(XMLExporter exporter) {
    super(new StandardSeedGenerator(), new
       FixedWaxmanNetworkGenerator(), true, exporter);

    //Set generator for resources
    mResGens.add(new FixedCpuResourceGenerator()); //needs
```

```
        parameters: Min_CPU and Max_CPU
    mResGens.add(new IdResourceGenerator());

    //Set generator for demands
    mDemGens.add(new FixedCpuDemandGenerator()); //needs
        parameters: Min_CPU and Max_CPU

    //Set Metrics
    mMetrics.add(new AcceptedVnrRatio());
    mMetrics.add(new RejectedNetworksNumber());
  }

  @Override
  protected IAlgorithm prepareRunnerStage2(TestRun tr) {
    NetworkStack ns = tr.getScenario().getNetworkStack();
    IAlgorithm algo = new SubgraphIsomorphismStackAlgorithm(ns,
        new AdvancedSubgraphIsomorphismAlgorithm(false));
    return algo;
  }
}
```

The super constructor will be called with the following parameters:

1. "SeedGenerator" based on `AbstractSeedGenerator`, can be null.
2. "NetworkGenerator" based on `AbstractNetworkGenerator`
3. `boolean` which says whether the links should be bidirectional or not
4. `XMLExporter` which will export the results

Then the generators for "Resources", "Demands" and the metrics need to be set. At last it's necessary to implement the method *prepareRunnerStage2()* which creates and prepares the algorithm which is used for the tests.

The last step is to run the tests, done by a simple main-class

```
public class ExampleMain {

  public static void main(String[] args) {
    //Create the TestGenerator and generate the TestSeries
    ExampleTestGenerator gen = new ExampleTestGenerator("
        Example␣TestSeries");
    TestSeries series = gen.generateTests();

    //Create the XMLExporter which is used, which doesn't
        export the Networks
    XMLExporter exporter = new XMLExporter("Result.xml", series
        .getTestSeriesName(),
    series.getTestGenerator(), false);

```

```
    //Create the TestRunner and run the tests in 5 parallel
       threads
    GeneratorTestRunner run = new GeneratorTestRunner(exporter)
       ;
    run.runAllTest(series.getAllTestRuns(), 5);
  }
}
```

5.3 Plotting

The class `Plotter` in package *plot* provides a simple method to filter the data and create plots in 2D and 3D. It's possible to filter the data using *applyFilter** methods. Each filter method need to have the following parameters:

1. Name of the parameter or metric as String
2. (Start) Value of the parameter with type `Double` or `String` (String only for *applyFilterEqual* and *applyFilterNotEqual*
3. End Value of the parameter with type `Double` (Only nneded in range filters)

If the statement is true, the test will removed from the list.

The output-methods always need to have a directory name, in which the results will be plotted and the files are named after the metric which is plotted. A 2D plot *output2d* need to have one parameter and a 3D plot *output3d* need to have two parameters. The ordering is based on the given parameter(s).
Be aware: At the moment it isn't possible to use `String` based parameters.

```
public class ExamplePlotMain {
  public static void main(String[] args) {
    TestSeries series2 = XMLImporter.importResults("Result.xml"
       );

    Plotter p = new Plotter(series2);
    p.applyFilterEquals("AcceptedVnrRatio", 0.0);

    p.output2D("ExamplePlot2d", "NumVNodesPerNet", "
       AcceptedVnrRatio");
    p.output3D("ExamplePlot3d", "NumVNodesPerNet", "Min_CPU", "
       AcceptedVnrRatio");
  }
}
```

5.4 Writing a Generator

To run tests there are different types of generators:

- "SeedGenerators" based on `AbstractSeedGenerator` which are creating seeds for random generators

- "NetworkGenerators" based on `AbstarctNetworkGenerator` which are creating the network stack
- "ResourceGenerators" based on `AbstarctResourceGenerator` which are creating a resource
- "DemandGenerators" based on `AbstarctResourceGenerator` which are creating a demand

A generator has always implement the methods *generate* which contains the logic and *reset* which resets the state if needed and is called before a every new `TestRun`. If the generator needs parameters, these are specified in the `GeneratorParameter` annotation. The parameter array will filled with these values in order of annotation. The following values are possible:

- "Seed:Seed": Seed from "SeedGenerator" from type `Double`
- "Networks:Networks": The `NetworkStack`
- "TR:ParameterName": The value for the parameter with the name "ParameterName" from `TestRun` (TR)
- "Result:package.ResoureOrDemandGenerator": The result of the *generate* method of the given and used demand or resource generator
- "Method:package.ResoureOrDemandGenerator|getFoo": The result of the non-static *getFoo* method of the given and in this "TestRunner"' used demand or resource generator
- "SMethod:some.package.Class|getBar": The result of the static *getBar* method of the given class "some.package.Class"

If a parameter is not available or there is an error, an `Error` is thrown. **Be aware:**Due to parallel execution of threads the call of static methods may lead to wrong results!

```
@GeneratorParameter(
    parameters = { "Networks:Networks", "TR:Min_CPU", "TR:
       Max_CPU", "Seed:Seed"}
    )
public class FixedCpuResourceGenerator extends
    AbstractResourceGenerator<List<CpuResource>> {

  @Override
  public List<CpuResource> generate(ArrayList<Object>
      parameters) {
    ArrayList<CpuResource> resList = new ArrayList<CpuResource
       >();

    NetworkStack ns = (NetworkStack)parameters.get(0);
    Integer minCPU = ConversionHelper.paramObjectToInteger(
       parameters.get(1));
```

```
    Integer maxCPU = ConversionHelper.paramObjectToInteger(
        parameters.get(2));
    Long seed = ConversionHelper.paramObjectToLong(parameters.
        get(3));

    Random random = new Random();
    random.setSeed(seed);

    SubstrateNetwork sn = ns.getSubstrate();

    for(SubstrateNode n : sn.getVertices()) {
      CpuResource cpu = new CpuResource(n);
      int value = (int) (minCPU + (maxCPU
          - minCPU + 1)
          * random.nextDouble());
      cpu.setCycles((double) value);
      n.add(cpu);
      resList.add(cpu);
    }
    return resList;
  }

  @Override
  public void reset() {}
}
```